ANTAREX: A DSL-based Approach to Adaptively Optimizing and Enforcing Extra-Functional Properties in High Performance Computing

Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, Joao M. R. Cardoso, Carlo Cavazzoni, et al.

ANTAREX: A DSL-based Approach to Adaptively Optimizing and Enforcing Extra-Functional Properties in High Performance Computing

Cristina Silvano*, Giovanni Agosta*, Andrea Bartolini†, Andrea R. Beccari‡, Luca Benini†, Loïc Besnard‡‡, João Bispo§, Radim Cmar††, João M. P. Cardoso§, Carlo Cavazzoni¶, Stefano Cherubin*, Davide Gadioli*, Martin Golasowski‖, Imane Lasri**, Jan Martinovič‖, Gianluca Palermo*, Pedro Pinto§, Erven Rohou**, Nico Sanna¶, Kateřina Slaninová‖, Emanuele Vitali*

*DEIB – Politecnico di Milano, †IIS – Eidgenössische Technische Hochschule Zürich,
‡Dompé Farmaceutici SpA, §FEUP – Universidade do Porto, ¶CINECA,
‖IT4Innovations, VSB – Technical University of Ostrava, **INRIA Rennes, ††Sygic, ‡‡IRISA/CNRS
Email: *name.surname@polimi.it †{barandre,lbenini}@iis.ee.ethz.ch, ‡andrea.beccari@dompe.it, §{jbispo,jmpc,pmsp}@fe.up.pt,
¶n.surname@cineca.it, ‖name.surname@vsb.cz, **name.surname@inria.fr, ††rcmar@sygic.com, ‡‡name.surname@irisa.fr

***Abstract*—The ANTAREX project relies on a Domain Specific Language (DSL) based on Aspect Oriented Programming (AOP) concepts to allow applications to enforce extra functional properties such as energy-efficiency and performance and to optimize Quality of Service (QoS) in an adaptive way. The DSL approach allows the definition of energy-efficiency, performance, and adaptivity strategies as well as their enforcement at runtime through application autotuning and resource and power management. In this paper, we present an overview of the ANTAREX DSL and some of its capabilities through a number of examples, including how the DSL is applied in the context of one of the project use cases.**

***Keywords*—High Performance Computing, Autotuning, Adaptivity, DSL, Compilers, Energy Efficiency**

I. Introduction

Designing and optimising applications for energy-efficient High Performance Computing systems up to the Exascale era is an extremely challenging problem. Exascale supercomputers (reaching billions of billions of floating point operations per second) cannot be built by simply expanding the number of processing nodes and leveraging technology scaling, as power demand would increase unsustainably (up to hundreds of MW). To reach the DARPA target of 20MW of Exascale supercomputers for the year 2020, current supercomputers (reaching up to 93 PetaFlop/s) must achieve an energy efficiency "quantum leap". The Green500 list [1] looks at the GigaFlops per Watt as energy efficiency metric to rank supercomputers by their energy efficiency. According to the latest Green500 list published in November 2017, the "most green" supercomputer SHOUBU SystemB installed in Japan achieves 17 GigaFlops/W during its 842-TeraFlop/s Linpack performance run. The top positions in Green500 are all occupied by heterogeneous systems based on high-performance processors and co-processors such as the latest NVIDIA Volta GV100 GPU and PEZY SC2 accelerator to further accelerate the computation. The dominance of heterogeneous systems in the Green500 list is expected to continue for the next coming years to reach the target of 20MW Exascale supercomputers. To this end, European efforts have recently been focused on building supercomputers out of the less power-hungry ARM cores and GPGPUs [1].

Designing and implementing HPC applications are difficult and complex tasks, which require mastering several specialized languages and tools for performance tuning. This is incompatible with the current trend to open HPC infrastructures to a wider range of users. The current model where the HPC center staff directly supports the development of applications will become unsustainable in the long term. Thus, the availability of effective standard programming languages and APIs is crucial to provide migration paths towards novel heterogeneous HPC platforms as well as to guarantee the ability of developers to work effectively on these platforms.

To fulfil the 20MW target, energy-efficient heterogeneous supercomputers need to be coupled with radically new software stacks to exploit the benefits offered by heterogeneity at all levels (supercomputer, job, node).

The ANTAREX [2, 3, 4] project aims at providing a holistic approach spanning all the decision layers composing the supercomputer software stack and exploiting effectively the full system capabilities, including heterogeneity and energy management. The main goal of ANTAREX is to express by means of a DSL the application self-adaptivity and to runtime manage and autotune applications for green heterogeneous HPC systems up to the Exascale level. The use of a DSL allows the introduction of a separation of concerns, where self-adaptivity and energy efficient strategies are specified separately from the application functionalities. The DSL is based on previous efforts regarding the LARA language [5, 6] and makes possible to express at compile time the adaptivity/energy/performance strategies and to enforce at runtime application autotuning and resource and power management. The goal is to support the parallelism, scalability and adaptivity in a dynamic workload by exploiting the full system capabilities (including energy management) for emerging large-scale and extreme-scale systems, while reducing the Total Cost of Ownership (TCO) for companies and public organizations.

The project is driven by two use cases taken from highly relevant HPC application scenarios: (1) a biopharmaceutical application for drug discovery deployed on the 1.21 PetaFlops heterogeneous NeXtScale Intel-based IBM system at CINECA and (2) a self-adaptive navigation system for smart cities deployed on the server-side on the 1.46 PetaFlops heterogeneous Intel® Xeon Phi™ based system provided by IT4Innovations National Supercomputing Center.

The ANTAREX Consortium comprises a wealth of expertise in all pertinent domains. Four top-ranked academic and research partners (Politecnico di Milano, ETHZ Zurich, University of Porto and INRIA) are complemented by the Italian Tier-0 Supercomputing Center (CINECA), the Tier-1 Czech National Supercomputing Center (IT4Innovations) and two industrial application providers, one of the leading biopharmaceutical companies in Europe (Dompé) and the top European navigation software company (Sygic). The complementarity and deep expertise of the Consortium partners has the potential to generate a breakthrough innovation from the ANTAREX project. Moreover, the presence of leading edge industrial partners will ensure

[1] www.green500.org, November 2017

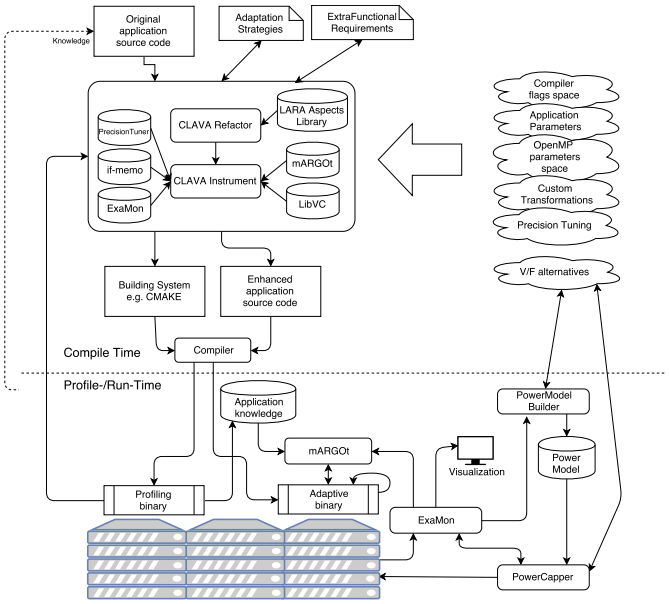

Fig. 1. The ANTAREX Tool Flow

a relevant impact on direct exploitation paths of ANTAREX results in industry and society. Politecnico di Milano, the largest Technical University in Italy, plays the role of Project Coordinator.

The ANTAREX approach and related tool flow, as shown in Figure 1, operate both at design-time and runtime. The application functionality is expressed through C/C++ code (possibly including legacy code), whereas the extra-functional aspects of the application, including parallelisation, mapping, and adaptivity strategies, are expressed through DSL code (based on LARA) developed in the project. As a result, the expression of such aspects is fully decoupled from the functional code. The *Clava* tool is the centerpoint of the compile-time phase, performing a refactoring of the application code based on the LARA aspects, and instrumenting it with the necessary calls to other components of the tool flow.

The ANTAREX compilation flow leverages a runtime phase with compilation steps, through the use of partial dynamic compilation techniques enabled by *libVC*. The application autotuning, performed via the *mARGOt* tool, is delayed to the runtime phase, where the software knobs (application parameters, code transformations and code variants) are configured according to the runtime information coming from application self-monitoring as well as from system monitoring performed by the *ExaMon* tool. Finally, the runtime power manager, *PowerCapper*, is used to control the resource usage for the underlying computing infrastructure given the changing conditions. At runtime, the application control code, thanks to the design-time phase, now contains also runtime monitoring and adaptivity strategy code derived from the DSL extra-functional specification. Thus, the application is continuously monitored to guarantee the required Service Level Agreement (SLA), while communication with the runtime resource-manager takes place to control the amount of processing resources needed by the application. The application monitoring and autotuning is supported by a runtime layer implementing an application level collect-analyse-decide-act loop.

The rest of this paper is organized as follows. In Section II we review the technology portfolio provided by the ANTAREX tool flow. In Section III we provide an assessment of the impact of the proposed DSL on application specifications, while Section IV gives an overview of how the Tool Flow has been used in one of the use cases. Finally, in Section V we draw some conclusions.

II. ANTAREX Technology Portfolio

A. The ANTAREX DSL

HPC applications might profit from adapting to operational and situational conditions, such as changes in contextual information (e.g., workloads), in requirements (e.g., deadlines, energy), and in availability of resources (e.g., connectivity, number of processor nodes available). A simplistic approach to both adaptation specification and implementation (see, e.g., [7]) employs hard coding of, e.g., conditional expressions and parameterizations. In our approach, the specification of runtime adaptability strategies relies on a DSL implementing key concepts from Aspect-Oriented Programming (AOP) [8], mainly specifying adaptation concerns, targeting specific execution points, separately from the primary functionality of the application, with minimum or no changes to the application source code.

Our approach is based on the idea that certain application/system requirements (e.g., target-dependent optimizations, adaptivity behavior and concerns) should be specified separately from the source code that defines the main functionality. Those requirements are expressed as DSL aspects that embody strategies. An extra compilation step, performed by a *weaver*, merges the original source code and aspects into the intended program [9]. Using aspects to separate concerns from the core objective of the program can result in cleaner programs and increased productivity (e.g., reusability of strategies). As the development process of HPC applications typically involves two types of experts (application-domain experts and HPC system architects) that split their responsibilities along the boundary of functional description and extra-functional aspects, our DSL-aided toolflow provides a suitable approach for helping to express their concerns.

The ANTAREX DSL relies on the already existing DSL technology LARA [5, 6]. In particular, the LARA technology provides a framework that we adopted to implement the ANTAREX aspects and APIs. Moreover, we developed other LARA-related tools such as the *Clava*[2] weaver to leverage the rest of the ANTAREX tool flow.

LARA is a programming language that allows developers to capture non-functional requirements and concerns in the form of strategies, which are decoupled from the functional description of the application. Compared to other approaches that usually focus on code injection (e.g., [10]), LARA provides access to other types of actions, e.g., code refactoring, compiler optimizations, and inclusion of additional information, all of which can guide compilers to generate more efficient implementations. Additional types of actions may be defined in the language specification and associated weaver, such as software/hardware partitioning [11] or compiler optimization sequences [12]. One important feature of the LARA-aided source-to-source compiler developed in ANTAREX is the capability to refactor the code of the application in order to expose adaptivity behavior and/or adaptivity design points that can be explored by the ANTAREX autotuning component. In the following sections we show illustrative examples[3] of some of the strategies that can be specified using LARA in the context of a source-to-source compiler and currently used for one of the use cases.

B. Precision Tuning

Error-tolerating applications are increasingly common in the emerging field of real-time HPC, allowing to trade-off precision for performance and/or energy. Thus, recent works investigated the use of customized precision in HPC as a way to provide a breakthrough

[2] https://github.com/specs-feup/clava

[3] Complete working versions for all examples can be found in https://github.com/specs-feup/specs-lara/tree/master/2018%20DSD

in power and performance. We developed a set of LARA aspects enabling mixed precision tuning on C/C++ and OpenCL kernels. In our precision tuning we combine an adaptive selection of floating and fixed point arithmetic, targeting HPC applications.

Figure 2 presents part of a LARA strategy that changes all declarations of a certain type to a target type (e.g., from double to float) for a given function. We note, however, that a practical and reusable aspect needs to deal with further issues, such as the cloning of functions whose types we want to change but are also called by other unrelated functions in the code, assignments of constants, casts, recursion, changing functions definitions and library functions to the ones related to the type used (e.g., `sqrtf` vs `sqrt` in Math.h), etc. In this example, `changeType` is a function that analyzes and changes compound types, such as `double*` and `double[]`. If the type described in `$old` is found inside the type of the declaration, it is replaced with the type described in `$new`. To be more specific, if `$old` is `double`, `$new` is `float` and `$decl.type` is `double*`, the type of the declaration will be changed to `float*`. If the original declaration type does not contain the `$old` type, it is not changed.

```
1  aspectdef ChangePrecision
2
3    input $func, $old, $new end
4
5    /* change type of variable declarations found
6     *  inside the function and parameters */
7    select $func.decl end
8    apply
9      var changedType = changeType($decl.type, $old, $new);
10     def type = changedType;
11   end
12
13   /* do the same with the function return type ... */
14   var $returnType = $func.functionType.returnType;
15   $func.functionType.def returnType =
16         changeType($returnType, $old, $new);
17 end
```

Fig. 2. Example of LARA aspect to change the types of variables declared inside a given function.

A LARA aspect consists of three main steps. Firstly, one captures the points of interest in the code using a `select` statement, which in this example selects variable declarations. Then, using the `apply` statement, one acts over the selected program points. In this case, it will define the types of the captured declared variables, using the `type` attribute. Finally, we can then specify conditions to constrain the execution of the `apply` (i.e., only if the declared variable has a specific type). This can be done via conditional statements (`if`s) as well as via special `condition` blocks that constrain the entire `apply`. LARA promotes modularity and aspect reuse, and supports embedding JavaScript code, to specify more sophisticated strategies. As shown in [13], we support exploration of mixed precision OpenCL kernels by using half, single, and double precision floating point data types. We additionally support fixed point representations through a custom C++ template-based implementation for HPC systems, which has already been used in [14]. In both cases the LARA aspects automatically insert code for proper type conversion before and after the critical section that has been converted to exploit a reduced precision data type.

The LARA aspect in Figure 3 shows the generation of different mixed-precision versions to be dynamically evaluated. It is possible to specify – as input of the aspect – the number of mix combination to generate, and a rule set to filter out precision mix combinations which are very likely to lead to useless and/or not efficient results. We exploit programmer's application domain knowledge by relying on them to define test cases to evaluate the different code versions at runtime. LARA automatically inserts code to dynamically perform the exploration over the space of the generated versions with different precision mix.

```
1  aspectdef HalfPrecisionOpenCL
2    input
3      combinationFilter = [],
4      maxVersions = undefined
5    end
6
7    // List of float and double vars in the OpenCL kernel
8    call result : OpenCLVariablesToTest;
9    var variablesToTest = result.variablesToTest;
10   // Sequence generator
11   var sequenceGenerator = new SequentialCombinations(
12     variablesToTest.length, maxVersions);
13   var counter = 0;
14   while(sequenceGenerator.hasNext()) {
15     Clava.pushAst(); // Save current AST
16
17     // Get a new combination of variables
18     var combination = sequenceGenerator.next();
19     var lastSeed = sequenceGenerator.getLastSeed();
20     if(!isCombinationValid(lastSeed, combinationFilter))
21       continue;
22     // Change the builtin type of the variables
23     for(var index of combination) {
24       var $vardecl = Clava.findJp(variablesToTest[index]);
25       changeTypeToHalf($vardecl);
26     }
27     call addHalfPragma(); // Enable half-precision
28     var outputFolder = createFolder(lastSeed,
29       variablesToTest.length, counter);
30     Clava.writeCode(outputFolder); // Generate code
31     Clava.popAst(); // Restore previous AST tree
32     counter++; // Increase counter
33   }
34 end
```

Fig. 3. Example of LARA aspect that generates different precision mix versions of the same OpenCL kernel.

C. Code Versioning

One of the strategies supported in the ANTAREX toolflow is the capability to generate versions of a function and to select the one that satisfies certain requirements at runtime. Figure 4 shows an aspect that clones a set of functions and changes the types of the newly generated clones. Each clone has the same name as the original with the addition of a provided suffix. We start with a single user-defined function which is cloned by the aspect `CloneFunction` (called in line 13). Then, it recursively traverses calls to other functions inside the clone and generates a clone for each of them. Inside the clones, calls to the original functions are changed to calls to the clones instead, building a new call tree with the generated clones. At the end of the aspect `CreateFloatVersion` (lines 16–17,) we use the previously defined `ChangePrecision` aspect to change the types of all generate clones.

The aspect `Multiversion` – in Figure 5 – adapts the source code of the application in order to call the original version of a function or a generated cloned version with a different type, according to the value of a parameter given by the autotuner at runtime. The main aspect calls the previously shown aspect, `CreateFloatVersion`, which clones the target function and every other function it uses, while also changing their variable types from `double` to `float` (using the aspects presented in Figure 4 and Figure 2). This is performed in lines 8–9 of the example. From lines 13 to 34, the `Multiversion` aspect generates and inserts code in the application that is used as switching mechanism between the two versions. It starts by declaring a variable to be used as a knob by the autotuner, then it generates the code for a switch statement and replaces the statement containing

```
import ChangePrecision;
import clava.ClavaJoinPoints;

aspectdef CreateFloatVersion
  input $func, suffix end
  output $clonedFunc end

  $double = ClavaJoinPoints.builtinType('double');
  $float = ClavaJoinPoints.builtinType('float');

  /* clone the target functions and the child calls */
  var clonedFuncs = {};
  call cloned : CloneFunction($func, suffix, clonedFuncs);

  /* change the precision of the cloned function */
  for($clonedFunc of clonedFuncs)
    call ChangePrecision($clonedFunc, $double, $float);

  $clonedFunc = cloned.$clonedFunc;
end
```

Fig. 4. Example of LARA aspect to clone an existing function and change the type of the clone.

the original call with the generated switch code. Finally, in lines 36–38, the aspect surrounds both calls (original and float version) with timing code. An excerpt of the resulting C code can bee seen in Figure 6.

```
import CreateFloatVersion;
import lara.code.Timer;
import clava.ClavaJoinPoints;

aspectdef Multiversion
  input $func, knobName end

  call fVersion : CreateFloatVersion($func, "_f");
  var $floatFunc = fVersion.$clonedFunc;
  var timer = new Timer();

   /* Identify call by name... */
   select function.body.stmt.call{$func.name} end
   apply
    /* ... and by type signature */
    if(!$func.functionType.equals($call.functionType))
      continue;

    /* Add knob for choosing the version */
    $int = ClavaJoinPoints.builtinType('int');
    $body.exec addLocal(knobName, $int, 0);

    /* create float declaration for first argument */
    var $arg = createFloatArg($call.args[0]);
    /* Create call based on float version of function */
    $floatFunc.exec $fCall : newCall([$arg, $call.args[1]]);
    /* Copy current call */
    $call.exec $callCopy : copy();

    /* Create switch */
    var $condition = ClavaJoinPoints.exprLiteral(knobName);
    var switchCases = {0: $callCopy, 1: $fCall};
    call switchJp : CreateSwitch($condition, switchCases);
    $stmt.exec replaceWith(switchJp.$switch);

    /* Time calls to both original and float functions*/
    timer.time($callCopy, "Original time:");
    timer.time($fCall, "Float time:");
   end
end
```

Fig. 5. Example of LARA aspect that generates an alternative version of a function and inserts a mechanism in the code to switch between versions.

In the ANTAREX toolflow, the capability of providing several versions of the same function is not limited to static features. LIBVERSIONINGCOMPILER [15] (abbreviated LIBVC) is an open-source C++ library designed to support the dynamic generation and

```
switch (version) {
  case 0: {
    clock_gettime(CLOCK_MONOTONIC, &time_start_0);
    SumOfInternalDistances(atoms, 1000);
    clock_gettime(CLOCK_MONOTONIC, &time_end_0);
    double time_0 = calc_time(time_start_0, time_end_0);
    printf("Original time:%fms\n", time_0);
  }
  break;
  case 1: {
    clock_gettime(CLOCK_MONOTONIC, &time_start_1);
    SumOfInternalDistances_f(atoms_f, 1000);
    clock_gettime(CLOCK_MONOTONIC, &time_end_1);
    double time_1 = calc_time(time_start_1, time_end_1);
    printf("Float time::%fms\n", time_1);
  }
  break;
}
```

Fig. 6. Excerpt of the C code resulting from the generation of alternative code versions.

versioning of multiple versions of the same compute kernel in a HPC scenario. It can be used to support continuous optimization, code specialization based on the input data or on workload changes, or to dynamically adjust the application, without the burden of a full just-in-time compiler. LIBVC allows a C/C++ compute kernel to be dynamically compiled multiple times while the program is running, so that different specialized versions of the code can be generated and invoked. Each specialized version can be versioned for later reuse. When the optimal parametrization of the compiler depends on the program workload, the ability to switch at runtime between different versions of the same code can provide significant benefits [16, 17]. While such versions can be generated statically in the general case, in HPC execution times can be so long that exhaustive profiling may not be feasible. LIBVC instead enables the exploration and tuning of the parameter space of the compiler at runtime.

Figure 7 shows an example of usage of LIBVC through LARA, which demonstrates how to specialize a function. The user provides this aspect with a target function call and a set of compilation options. These include compiler flags and possible compiler definitions, e.g., data discovered at runtime, which is used as a compile-time constant in the new version. Based on the target function call, the aspect finds the function definition which is passed to the library. After the options are set, the original function call is replaced with a call of the newly compiled and loaded specialized version of the kernel.

```
import antarex.libvc.LibVC;

aspectdef SimpleLibVC

  input
    name, $target, options
  end

  var $function = $target.definition;
  var lvc = new LibVC($function, {logFile:"log.txt"}, name);

  var lvcOptions = new LibVCOptions();
  for (var o of options) {
    lvcOptions.addOptionLiteral(o.name, o.value, o.value);
  }
  lvc.setOptions(lvcOptions);

  lvc.setErrorStrategyExit();

  lvc.replaceCall($target);
end
```

Fig. 7. Example of LARA aspect to replace a function call to a kernel with a call to a dynamically generated version of that kernel.

It is worth noting that the combination of LARA and LIBVC can also be used to support compiler flag selection and phase-ordering both statically and dynamically [18, 19].

D. Memoization

Optimising applications for energy-efficiency is a challenge of the ANTAREX project. We introduce in this section a memoization technique integrated in the ANTAREX toolflow. Performance can be improved by caching results of pure functions (i.e. deterministic functions without side effects), and retrieving them instead of recomputing a result. We have implemented the work of [20] generalized for C++ and aided with extensions regarding user/developer flexibility. We describe here only the principles of this technique and more details can be found in [21] [20].

```
1   float foo (float p) {
2   /* code of foo without side effects */
3   }
4
5   float foo_wrapper(float p)
6       {
7         float r;
8         /* already in the table ? */
9         if (lookup_table(p, &r)) return r;
10        /* calling the original function */
11        r = foo(p);
12        /* updating the table or not */
13        update_table(p, r);
14        return r;
```

Fig. 8. A memoizable C function and its wrapper.

Consider a memoizable C function `foo` as shown in Figure 8. The memoization consists in:

1) the insertion of a wrapper function `foo_wrapper` and an associated table.
2) The substitution of the references to `foo` by `foo_wrapper` in the application.

This technique has been extended for C++ memoizable methods and takes into account the mangling, the overloading, and the references to the objects. Memoization is proposed in the ANTAREX project by relying on aspects programmed using the DSL. The advantage of these aspects is that the memoization is integrated into the application without requiring user modifications of the source code. The code generated by Clava is then compiled and linked with the associated generated memoization library.

An example of a LARA aspect for memoization is shown in Figure 9. It defines the memoization (lines 1-13) of a method (`aMethod`) of a class (`aClass`) with `nbArg` parameters of same type as the returned type (`Type`). Note that the inputs `nbArg` and `Type` are required to manage the overloading of the object-oriented languages such that C++. Other parameters (from line 4) are provided to improve several memoization approaches. For examples, the user can specify (1) the policy in case of conflicts regarding the same table entry (line 11): replacement or not in case of conflict to the same entry of the table for different parameters of the memoized function, and (2) the size of the table (line 15). After some verifications, not detailed here, on the parameters (lines 14-15), the method is searched (lines 17-24). Then, in case of success, the code of the wrapper is added (line 28) to produce the memoization library, and (line 30) the code of the application is modified for calling the created "wrapper", this wrapper is also declared as a new method of the class.

Moreover, some variables are exposed for autotuning in the memoization library. For each function or method, a variable that manages the dynamical "stop/run" of the memoization is exposed, as well as

```
1  aspectdef Memoize_Method_overloading_ARGS
2  input
3    aClass,      // Name of a class
4    aMethod,     // Name of a method of the class aClass
5    pType,       // Name of the selected type
6    nbArgs,      // Number of parameters of the method
7    fileToLoad, // filename for init of the table, or 'none'
8    FullOffLine,// yes for a fully offline strategy
9    FileToSave, // filemane to save the table, or 'none'
10   Replace,     // Always replace in case of collisions
11   approx,      // Number of bits to delete for approximation.
12   tsize        // Size of the internal table.
13 end
14 // Control on the parameters of the aspect: nbArgs in [1,3]
15    ...
16 // Searching the method.
17 var MethodToMemoize, found=false;
18 select class{aClass}.method{aMethod} end
19 apply
20  if (! found) {
21    found = isTheSelectedMethod($method, nbArgs, pType);
22    if (found) MethodToMemoize=$method;
23  }
24 end
25 if (!found)
26  { /* message to the user */}
27 else {
28  GenCode_CPP_Memoization(aClass, aMethod, pType, nbArgs,
29  fileToLoad, FullOffLine, FileToSave, Replace, approx, tsize);
30  call CPP_UpdateCallMemoization(aClass, aMethod, pType, nbArgs);
31 }
32 end
```

Fig. 9. An example of LARA aspect defined for the memoization.

the variable that manages the policy to use in case of conflict to the table. To be complete about the memoization, a LARA aspect is proposed to automatically detect the memoizable functions or methods. Then the user may decide to apply or not the memoization on these selected elements.

E. Self-Adaptivity & Autotuning

In ANTAREX, we consider each application's function as a parametric function that elaborates input data to produce an output (i.e., $o = f(i, k_1, \ldots, k_n)$), with associated extra-functional requirements. In this context, the parameters of the function $(k_1, \ldots, k_n)$ are software-knobs that modify the behavior of the application (e.g., parallelism level or the number of trials in a MonteCarlo simulation). The main goal of mARGOt [4] [22] is to enhance an application with an adaptive layer, aiming at tuning the software knobs to satisfy the application requirements at runtime. To achieve this goal, the mARGOt dynamic autotuning framework developed in ANTAREX is based on the MAPE-K feedback loop [23]. In particular, it relies on an application knowledge, derived either at deploy time or at runtime, that states the expected behavior of the extra-functional properties of interest. To adapt, on one hand mARGOt uses runtime observations as feedback information for reacting to the evolution of the execution context. On the other hand, it considers features of the actual input to adapt in a more proactive fashion. Moreover, the framework is designed to be flexible, defining the application requirements as a multi-objective constrained optimization problem that might change at runtime.

To hide the complexity of the application enhancement, we use LARA aspects for configuring mARGOt and for instrumenting the code with related API. Figure 10 provides a simple example of a LARA aspect where mARGOt has been configured (lines 5-20) to actuate on a software knob *Knob1* and target *error* and *throughput* metrics [24]. In particular, the optimization problem has been defined

[4] https://gitlab.com/margot_project/core

as the maximization of the *throughput* while keeping the *error* under a certain threshold. The last part of the aspect (lines 23-27) is devoted to the actual code enhancement including the needed mARGOt call for initializing the framework and for updating the application configuration. The declarative nature of the LARA library developed for integrating mARGOt simplifies its usage hiding all the details of the framework.

```
1  aspectdef mARGOt_Aspect
2  /* Input: TargetFunctionCall*/
3  input targetCallName end
4  /* mARGOt configuration */
5    var config = new MargotConfig();
6    var targetBlock = config.newBlock($targetCallName);
7    targetBlock.addKnob('Knob1', 'knob1', 'int');
8    targetBlock.addMetric('error', 'float');
9    targetBlock.addMetric('throughput', 'float');
10   targetBlock.addMetricGoal('my_error_goal',
11     MargotCFun.LE, 0.03, 'error');
12
13   /* optimization problem */
14   var problem = targetBlock.newState('defaultState');
15   problem.maximizeMetric('throughput');
16   problem.subjectTo('my_error_goal');
17
18   /* generate the information needed
19      for enhancing the application code */
20   codegen = MargotCodeGen.fromConfig(config, $targetCallName);
21
22 /* Target function call identification */
23 select stmt.call{targetName} end
24
25 /* Add mARGOt calls*/
26 codegen.init($call);
27 codegen.update($call);
28 end
```

Fig. 10. Example of a LARA aspect for autotuner configuration and code enhancement.

F. Monitoring

Today processing elements embed the capability of monitoring their current performance efficiency by inspecting the utilization of the micro-architectural components as well as a set of physical parameters (i.e., power consumption, temperature, etc). These metrics are accessible through hardware performance counters which in x86 systems can be read by privilege users, thus creating practical problems for user-space libraries to access them. Moreover, in addition to sensors which can be read directly from the software running on the core itself, supercomputing machines embed sensors external to the computing elements but relevant to the overall energy-efficiency. These elements include the node and rack cooling components as well as environmental parameters such as the room and ambient temperature. In ANTAREX, we developed ExaMon[25] (*Exa*scale *Mon*itoring) to virtualise the performance and power monitoring access in a distributed environment. ExaMon decouples the sensor readings from the sensor value usage. Indeed, ExaMon uses a scalable approach were each sensor is associated to a sensing agent which periodically collects the metrics and sends the measured value with a synchronized time-stamp to an external data broker. The data broker organises the incoming data in communication channels with an associated topic. Every new message on a specific topic is then broadcast to the related subscribers, according to a list kept by the broker. The subscriber registers a callback function to the given topic which is called every time a new message is received. To let LARA take advantage of this monitoring mechanism we have designed the Collector API, which allow the initialization of the Collector component associated with a specific topic that keeps an internal state of the remote sensor updated. This internal state can then be queried asynchronously by the Collector API to gather its value. LARA aspects have been designed to embed the Collector API and to make the application code self-aware.

Figure 11 shows a usage example of ExaMon through LARA, which subscribes to a topic on a given broker and inserts a logging message in the application. To define the connection information, the user needs to provide the address to connect to, as well as the name of the topic to subscribe. As for the integration in the original application code the user needs to provide a target function, where the collector will be managed, and a target statement, where the query of the data and logging will be performed.

```
1  import antarex.examon.Examon;
2  import lara.code.Logger;
3
4  aspectdef SimpleExamon
5
6    input
7      name, ip, topic, $manageFunction, $targetStmt
8    end
9
10   var broker = new ExamonBroker(ip);
11
12   var exa = new ExamonCollector(name, topic);
13
14   // manage the collector on the target function
15   select $manageFunction.body end
16   apply
17     exa.init(broker, $body);
18     exa.start($body);
19
20     exa.end($body);
21     exa.clean($body);
22   end
23
24   // get the value and use it in the target stmt
25   exa.get($targetStmt);
26
27   // get the last stmt of the scope of the target stmt
28   var $lastStmt = $targetStmt.ancestor("scope").lastStmt;
29   // Create printf for time and data
30   var logger = new Logger();
31   logger.ln().text("Time=").double(getTimeExpr(exa))
32     .text("[s], data=").double(exa.getMean()).ln();
33   // Add printf after last stmt
34   logger.log($lastStmt);
35 end
```

Fig. 11. Example of a LARA aspect integrate an ExaMon collector into an application.

G. Power Capping

Today's computing elements and nodes are power limited. For this reason, state-of-the-art processing elements embed the capability of fine-tuning their performance to control dynamically their power consumption. This includes dynamic scaling of voltage and frequency, and power gating for the main architectural blocks of the processing elements, but also some feedback control logic to keep the total power consumption of the processing element within a safe power budget. This logic in x86 systems is named RAPL [26]. Demanding the power control of the processing element entirely to RAPL may not be the best choice. Indeed, it has been recently discovered that RAPL is application agnostic and thus tends to waist power in application phases which exhibit IOs or memory slacks. Under these circumstances there are operating points that proved to be more energy efficient than the ones selected by RAPL while still respecting the same power budget [27]. However, these are only viable if the power capping logic is aware of the application requirements. To do so, we have developed a new power capping run-time based on a set of user space APIs which can be used to define a relative priority for the given task currently in execution on a given core. Thanks to

this priority, the run-time is capable of allocating more power to the higher priority process [28, 29]. In ANTAREX, these APIs can be inserted by LARA aspects in the application code.

III. Evaluation

Tables I and II show static and dynamic metrics collected for the weaving process of the presented examples. In Table I, we can see the number of logical lines of source code for the LARA strategies, as well as for the input code and generated output code (the SLoC-L columns). In the last two columns we report the difference in SLoC and functions between the input and output code (the delta columns). Note the woven and delta results for the HalfPrecisionOpenCL strategy are the sum of all generated code, totaling 31 versions.

An inspection of columns LARA SLoC-L and Delta SLoC-L reveals that, in most examples, there is a large overhead in terms of LARA SLoC-L over application SLoC-L. While this may seem a problem, we need to consider that a large part of the work being performed by these strategies is code analysis, which does not translate directly to SLoC-L in the final application. Furthermore, the Delta SLoC-L metric does not account for removed application code and for these cases a metric based on the similarity degree among code versions could be of more interest. Also, in real-world applications, the ratio of LARA SLoC-L to application SLoC-L would be definitely more favorable, thanks to aspect reuse.

To better understand the impact of analysis, we report in the first two columns of Table II the number of code points and of their attributes analysed, which can be compared with the last three column of the same table, which instead report the corresponding effects, in terms of the number of modified points and lines of code inserted. To understand the impact of removed lines of code, we look at the *Inserts* and *Actions* columns, which show that circa one half of the actions do not insert code. The end line is that the analysis work exceeds the transformation work by an order of magnitude, and the insertions only underestimate significantly the work performed.

Another benefit for user productivity when using LARA is how the techniques presented in the examples can scale into large-scale applications and scenarios. Most of the presented strategies are parameterized by function, i.e., they receive a function join point or name and act on the corresponding function. This could be performed manually, albeit crudely, using a search function of an IDE. Consider the case where we instead want to target a set of functions, whose names we may not know, based on their function signatures, or based on the characteristics of the variables declared inside their scope. This kind of search and filtering based on syntactic and semantic information available in the program is one of the key features of LARA and it cannot be easily attained with other tools. As the aspects presented here illustrate, LARA strategies can be made reusable and applied over large applications, greatly out scaling the effort needed to develop them.

IV. Case Study: Self-Adaptive Navigation System

In this section, we provide an overview of the application of the ANTAREX tool flow to the Self-Adaptive Navigation System developed in Use Case 2. The system is designed to process large volumes of data for the global view computation and to handle dynamic loads represented by incoming routing requests from users of the system. Both disciplines require HPC infrastructure in order to operate efficiently while maintaining contracted SLA. Integration of the ANTAREX self-adaptive holistic approach can help the system to meet the mentioned requirements and pave the way to scaling its operation to future Exascale systems.

Core of the system is a routing pipeline with several stages which uses our custom algorithm library written in C++. The library provides an API for the individual routing algorithms and for data access layer, which provides abstraction of a graph representation of the road network. The graph is stored in a HDF5 file, which is a well known and convenient storage format for structured data on HPC clusters.

As an example, we are using LARA aspects to generate C++ code for mapping native data types to types defined by the HDF5 API. The aspects are applied using the *Clava* tool which is a C++ frontend for the LARA toolchain. The *Clava* tool is integrated in our CMake-based build process as a custom build step, which parses the C++ structures representing the routing graph in memory and produces part of the HDF5 data access API. Details of the implementation can be found in [30]. Using the same process, other LARA aspects can be easily applied on the source code of the library, which greatly simplifies integration of other tools of the ANTAREX toolchain, such as mARGOt [22].

Furthermore, the mARGOt [22] autotuner is used in the Probabilistic Time-Dependent routing (PTDR) algorithm [31] to dynamically adjust the number of Monte Carlo samples used for the particular routing request. This parameter directly affects load generated by the PTDR stage and precision of its output. The autotuner uses operation point lists generated by a Design Space Exploration phase. The operation points in the context of PTDR are represented by a number of MC samples as an adjustable algorithm parameter and expected values of various metrics. The autotuner then dynamically selects the operation point according to the current request input. This approach can significantly reduce computational load generated by the PTDR phase, contributing to the overall efficient operation of the system.

Currently, our codebase is ready to use the DSL to integrate other tools from the ANTAREX tool flow. The autotuner is manually integrated in the routing pipeline, while verification of its correct function is ongoing. The next step is to use LARA to integrate the autotuner to the target application and evaluate its impact.

We have developed a server-side routing dashboard web application which is used to monitor the current status of the routing service. The application also provides a consistent environment for testing the service performance. It provides a way to execute a benchmark of the service by adjusting its parameters and sending a pre-defined set of routing requests. The service performance is then measured and results of the testing are stored for further analysis. The application also provides a consistent visualisation of the results which can be used for further analysis. This infrastructure will be used for validation of the ANTAREX tools integrated in the routing service [32].

V. Conclusions

To fully exploit the heterogeneous resources of future Exascale HPC systems, new software stacks are needed to provide power management, optimization, and autotuning to the parallel applications deployed on such systems. The ANTAREX project provides a holistic system-wide adaptive approach for next generation HPC systems, centered around a domain specific language that allows a full decoupling of functional and extra-functional specifications for each application, providing integration with a wide range of support tools. We have shown how the ANTAREX tool flow allows developers to control the precision of a computation, to manage dynamic code specialization, monitoring, power capping, and dynamic autotuning. The impact and benefits of such technology are far reaching, beyond traditional HPC domains.

Acknowledgements

The ANTAREX project is supported by the EU H2020 FET-HPC program under grant 671623.

TABLE I
STATIC METRICS

Strategy	LARA SLoC-L	LARA Aspects	Input SLoC-L	Input Func	Woven SLoC-L	Woven Func	Delta SLoC-L	Delta Func
ChangePrecision	27	1	12	3	13	3	1	0
SimpleExamon	20	1	12	3	23	5	11	2
Multiversion	46	2	12	3	43	5	31	2
CreateFloatVersion	28	2	12	3	24	3	12	0
SimpleLibVC	12	1	12	3	39	4	27	1
HalfPrecisionOpenCL*	93	3	9	1	279	31	270	30
Total	226	10	69	16	421	51	352	35

TABLE II
DYNAMIC METRICS

File	Selects	Attributes	Actions	Inserts	Native SLoC
ChangePrecision	4	109	2	1	0
SimpleExamon	4	131	18	7	0
Multiversion	8	477	27	16	9
HalfPrecisionOpenCL	125	2211	381	159	31
CreateFloatVersion	2	170	6	3	0
SimpleLibVC	7	93	13	8	36
Total	150	3191	447	194	76

REFERENCES

[1] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?" in *Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 40:1–40:12.

[2] C. Silvano, G. Agosta *et al.*, "AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems: the ANTAREX Approach," in *Design, Automation, and Test in Europe*, Dresden, Germany, Mar. 2016.

[3] C. Silvano, G. Agosta, S. Cherubin *et al.*, "The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems," in *2016 ACM Int'l Conf on Computing Frontiers*, 2016, pp. 288–293.

[4] C. Silvano, A. Bartolini, A. Beccari, C. Manelfi, C. Cavazzoni, D. Gadioli, E. Rohou, G. Palermo, G. Agosta, J. Martinovič *et al.*, "The ANTAREX Tool Flow for Monitoring and Autotuning Energy Efficient HPC Systems (Invited paper)," in *SAMOS 2017-Int'l Conf on Embedded Computer Systems: Architecture, Modeling and Simulation*, 2017.

[5] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: An Aspect-oriented Programming Language for Embedded Systems," in *Proc. 11th Annual Int'l Conf. on Aspect-oriented Software Development*. ACM, 2012, pp. 179–190.

[6] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonćalves, "Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach," *Software: Practice and Experience*, Dec. 2014.

[7] J. Floch *et al.*, "Using architecture models for runtime adaptability," *IEEE Softw.*, vol. 23, no. 2, pp. 62–70, Mar. 2006.

[8] J. Irwin, G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, and J.-M. Loingtier, "Aspect-oriented Programming," in *ECOOP'97 – Object-Oriented Programming*, ser. LNCS. Springer, 1997, vol. 1241, pp. 220–242.

[9] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented Programming: Introduction," *Commun ACM*, vol. 44, no. 10, pp. 29–32, 2001.

[10] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-oriented Extension to the C++ Programming Language," in *Proc. 40th Int'l Conf on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, 2002, pp. 53–60.

[11] J. M. Cardoso, T. Carvalho, J. G. Coutinho, R. Nobre, R. Nane, P. C. Diniz, Z. Petrov, W. Luk, and K. Bertels, "Controlling a complete hardware synthesis toolchain with LARA aspects," *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1073–1089, 2013.

[12] R. Nobre, L. G. Martins, and J. M. Cardoso, "Use of Previously Acquired Positioning of Optimizations for Phase Ordering Exploration," in *Proc. of Int'l Workshop on Software and Compilers for Embedded Systems*. ACM, 2015, pp. 58–67.

[13] R. Nobre, L. Reis, J. Bispo, T. Carvalho, J. M. P. Cardoso, S. Cherubin, and G. Agosta, "Aspect-driven mixed-precision tuning targeting gpus," in *PARMA-DITAM '18*, Jan 2018.

[14] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys, "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error," in *Int'l Conf on Parallel Computing (ParCo)*, Sep 2017.

[15] S. Cherubin and G. Agosta, "libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions," *SoftwareX*, vol. 7, pp. 95 – 100, 2018.

[16] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 datasets," in *Proc 31st ACM SIGPLAN Conf on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2010, pp. 448–459.

[17] M. Tartara and S. Crespi Reghizzi, "Continuous learning of compiler heuristics," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 46:1–46:25, Jan. 2013.

[18] A. H. Ashouri, G. Mariani *et al.*, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 21:1–21:25, Jun. 2016.

[19] A. H. Ashouri, A. Bignoli *et al.*, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 29:1–29:28, Sep. 2017.

[20] A. Suresh, E. Rohou, and A. Seznec, "Compile-Time Function Memoization," in *26th International Conference on Compiler Construction*, Austin, United States, Feb. 2017.

[21] A. Suresh *et al.*, "Intercepting Functions for Memoization: A Case Study Using Transcendental Functions," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, p. 23, Jul. 2015.

[22] D. Gadioli, G. Palermo, and C. Silvano, "Application autotuning to support runtime adaptivity in multicore architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 Int'l Conf on*. IEEE, 2015, pp. 173–180.

[23] Y. Brun *et al.*, *Engineering Self-Adaptive Systems through Feedback Loops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_3

[24] D. Gadioli *et al.*, "Socrates - a seamless online compiler and system runtime autotuning framework for energy-aware applications," in *Proc. of Design Automation and Test in Europe (DATE18)*, 2018, pp. 1149–1152.

[25] F. Beneventi, A. Bartolini, C. Cavazzoni, and L. Benini, "Continuous learning of hpc infrastructure models using big data analytics and in-memory processing tools," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 1038–1043.

[26] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Aug 2010, pp. 189–194.

[27] D. Cesarini, A. Bartolini, and L. Benini, "Benefits in relaxing the power capping constraint," in *ANDARE '17*. ACM, 2017, pp. 3:1–3:6.

[28] A. Bartolini, R. Diversi, D. Cesarini, and F. Beneventi, "Self-aware thermal management for high performance computing processors," *IEEE Design & Test*, 2017.

[29] D. Cesarini, A. Bartolini, and L. Benini, "Prediction horizon vs. efficiency of optimal dynamic thermal control policies in hpc nodes," in *2017 IFIP/IEEE Int'l Conf on Very Large Scale Integration (VLSI-SoC)*, Oct 2017, pp. 1–6.

[30] M. Golasowski, J. Bispo, J. Martinovič, K. Slaninová, and J. M. Cardoso, "Expressing and applying c++ code transformations for the hdf5 api through a dsl," in *IFIP Int'l Conf on Computer Information Systems and Industrial Management*. Springer, 2017, pp. 303–314.

[31] M. Golasowski, R. Tomis, J. Martinovič, K. Slaninová, and L. Rapant, "Performance evaluation of probabilistic time-dependent travel time computation," in *IFIP Int'l Conf on Computer Information Systems and Industrial Management*. Springer, 2016, pp. 377–388.

[32] M. K. L. Zadnik Stirn, Ed., *Server-side navigation service benchmarking tool*, Slovenian Society Informatika. Litostrojska cesta 54, Ljublana, Slovenia: Slovenian Society Informatika, Sept 2017.